Assignment 5 – Towers Report Template

Arushi Tyagi

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.
The purpose of this assignment is to create a hash table using linked lists and key-value pairs. We will implement and manage memory in these data structures and also optimize the hash table. Then, we will also use the hash table to count how many unique lines are given in an input.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- In Part I, you implemented garbage collection—routines that clean up dynamically-allocated memory. How did you make sure the memory was all cleaned up? How did you check?
  As it goes through the list, it frees the node. If the removed node is not the first node (another node was added to the beginning), it'll skip over it when redefining the current pointer and then remove that node. That way, every node is removed. When destroying the whole list, it continuously increments the node and frees it until the list is empty. It then frees the memory allocation of the list as well.
- In Part II, you made a major optimization to the linked list optimization. What was it, and why do you think it changed the performance of bench1 so much?
  Instead of adding to the end of the list, the program adds nodes to the beginning of the list so that it doesn't have to traverse through the list and then add it at the end. This makes it more efficient because it saves the amount of time that it takes to go through the list an n number of times.
- In Part III, you implemented hash tables. What happens to the performance of bench2 as you vary the number of buckets in your hash table? How many buckets did you ultimately choose to use?
  As the number of buckets increases, the performance of bench2 is better and faster. I chose 1000 buckets because it was a large amount but didn't take up too much memory.
- How did you make sure that your code was free from bugs? What did you test, and how did you test it? In particular, how did you create inputs and check the output of uniqq?
  I put in an input of multiple unique lines. I also put in an input of multiple lines with some repeating ones to make sure that the program only counted the unique lines and didn't count the same one multiple times.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays and a wide range of files/characters.

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

Testing for uniqq:

- Testing no inputs (output: 0)
- Testing multiple new lines (output: 1)
- Testing one input multiple times. (output: 1)
- Testing one unique input and two of the same ones. 3 lines in total. (output: 2)

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.
In order to test how fast the hashtables are, in the command prompt type: $ time bench1 to test bench1. For bench2, type: $ time bench2. In order to do multiple iterations of it and test how fast it is, type: $ time bench1 100 (if you want 100 iterations). Similarly, for bench 2 it is: $ time bench2 100. In order to run uniqq which would use the hash table implementation to calculate how many unique lines there are in the given input the user would type compile and run uniqq.c. Then, they would enter in as many lines as they want separated with a new line. To finish inputting text, hit control D and the program will output the numebr of unique lines from the standard input and the program will terminate.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.
In the linked list implementation, there is an LL structure for the linked lists. There are nodes that contains the data which is associated with the key-value pair. In this case, it is an item structure that holds the string which is the key and the id which is the value. Then, in the hash tables, there are buckets that use the linked lists to store the key-value pairs which map to the individual bucket.
How the linked list works is that a new node is added to the beginning of the list when a new key-value pair is being added. When looking for the specific key, we go through the list to find the bucket which has the pointer to the key. To remove a key, we have to find the node that contains the matching key and then remove the node from the linked list.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.
ll.c: works with linked lists.

- First, we need to allocate memory for the LL structure. Check if it is empty. If it is then the head pointer will be null. Return the head pointer.
- In order to add a node to the list, we will be adding it to the beginning for optimization. Allocate memory for the new node that holds item i. If allocation fails return false. Otherwise, copy item's data into field of node. (n-¿data = *i) Initialize n-¿next as NULL. If the list is empty, then the head of the list is the added node. If we are adding nodes to an existing list, the next node pointer is the head of the list and the head pointer of LL is the new added node.

- When trying to find a node in the list, start at the head of the linked list. While there is a node in the linked list, compare the current node with an item. If it matches then return the pointer of the item and go to the next node to do the same.
- To remove a node from the list, initialize the current node pointer to the head of the list and the create a previous pointer as well. While the list is not empty, check if the node and item that you are trying to remove match. If it is the first node, update the head pointer to the next node. If it doesn't, skip the removed node when redefining the previous pointer. Free the memory of the current node and increment the previous and current pointers.
- To destroy the whole linked list, while the linked list is not empty, free the current node and continue to increment it. Once it is empty, free the memory of the linked list and set the pointer to NULL.

hash.c: works with the hash table implementation

- In order to create the hash table, we have to allocate the memory and pointer of it and check for allocation failure. The size of the hash table would be the number of buckets. Allocate memory for the array of LL* pointers.
- to put a key-value pair in the hash table, sum the ASCII values of each character. Then divide it with the bucket count to get the remainder which is the bucket index for the key. If bucket index doesn't exist, then create a new linked list and store it in the buckets array. Then, create an item structure and copy its key string into a key. Then put its id into the value variable. Check if the item with the same key exists in the linked list. If it does, they remove the item with the same key and add the final item with the new value. Else, add item into the bucket because it is a new key.
- Get the value associated with the key. Similar to the last part, calculate the sum of the ASCII values of the key and then calculate the bucket index using modulus. Look for the item at the bucket index and return the id of the item that holds its value.
- Finally, destroy the hash table iterate through the buckets and if there is a linked list in the current bucket, destroy the list. Free the memory of the array of pointers, hash table structure, and set the pointer to NULL.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

ll.c

- Function: list_create. Inputs: none. Outputs: head of list pointer. Purpose: creates a linked list.
- Function: list_add. Inputs: linked list pointer, item pointer. Outputs: boolean. Purpose: adds a node to the linked list.
- Function: list_find. Inputs: linked list pointer, comparison of items boolean, item pointer. Outputs: item pointer. Purpose: finds the node in the linked list.
- Function: list_remove. Inputs: linked list pointer, comparison of items boolean, item pointer. Outputs: none. Purpose: removes item from linked list using cmp.
- Function: list_destroy. Inputs: linked list pointer. Outputs: none. Purpose: destroys linked list.

hash.c

- Function: hash_create. Inputs: none. Outputs: hash table pointer. Purpose: creates a hash table.

- Function: hash_put. Inputs: hash table pointer, key pointer, value. Outputs: boolean. Purpose: adds a key-value pair to the hash table.
- Function: hash_get. Inputs: linked list pointer, comparison of items boolean, item pointer. Outputs: item pointer. Purpose: finds the node in the linked list.
- Function: hash_destroy. Inputs: array of hash table pointers. Outputs: none. Purpose: destroys the hash table.

item.c

- Function: cmp. Inputs: pointer to item 1, pointer to item 2. Output: boolean true or false. Purpose: compares the keys of two items.

uniqq.c

- Function: main. Inputs: none. Outputs: number of unique lines. Purpose: takes in a file input from stdin. Counts and prints the number of unique lines.

References